Dimensions and Principles of Declassification

Andrei Sabelfeld David Sands
Chalmers*

Abstract

Computing systems often deliberately release (or declassify) sensitive information. A principal security concern for systems permitting information release is whether this release is safe: is it possible that the attacker compromises the information release mechanism and extracts more secret information than intended? While the security community has recognised the importance of the problem, the state-of-the-art in information release is, unfortunately, a number of approaches with somewhat unconnected semantic goals. We provide a road map of the main directions of current research, by classifying the basic goals according to what *information is released,* who *releases information,* where *in the system information is released, and* when *information can be released. With a general declassification framework as a long-term goal, we identify some prudent* principles *of declassification. These principles shed light on existing definitions and may also serve as useful "sanity checks" for emerging models.*

1. Introduction

Computing systems often deliberately release (i.e, *declassify* or *downgrade*) sensitive information. Without a possibility to leak secrets, some systems would be of no practical use. For example, releasing the average salary from a secret database of salaries is sometimes needed for statistical purposes. Another example of deliberate information release is information purchase. An information purchase protocol reveals the secret information once a condition (such as "payment transferred") has been fulfilled. Yet another example is a password checking program that leaks some information about the password. Some information is released even if a log-in attempt fails: the attacker learns that the attempted sequence is *not* the same as the password.

* Department of Computer Science, Chalmers University of Technology and the University of Göteborg, 412 96 Göteborg, Sweden. `www.cs.chalmers.se/~{andrei,dave}`

In *Proceedings of the IEEE Computer Security Foundations Workshop*, Aix-en-Provence, France, June 20-22, 2005.

Information release is a necessity in these scenarios. However, a principal security concern for systems permitting information release is whether this release is safe. In other words, is it possible that the attacker compromises the mechanism for information release and extracts more secret information than intended? Applying this question to the examples above: can individual salaries be (accidentally or maliciously) released to the attacker in the average salary computation? Can the attacker break an information purchase protocol to extract sensitive information before the payment is transferred? Is it possible that along with the result of password matching some other secret information is sneaked to the attacker? This leads to the following general problem:

> What are the policies for expressing intentional information release by programs?

Answering this questions is a crucial challenge [SM03, Zda04] for information security. Because many systems rely on information release, we believe that answering this question satisfactorily is the key to enabling technology transfer from existing information security research into standard security practice.

While the security research community has recognised the importance of the problem, the state-of-the-art in information release comprises a fast growing number of definitions and analyses for different kinds of information release policies over a variety of languages and calculi. The relationship between different definitions of release is often unclear and, in our opinion, the relationships that do exist between methods are often inaccurately portrayed. This creates hazardous situations where policies provide only partial assurance that information release mechanisms cannot be compromised.

For example, consider a policy for describing *what* information is released. This policy stipulates that at most four digits of a credit card number might be released when a purchase is made (as often needed for logging purposes). This policy specifies *what* can be released but says nothing about *who* controls which of the numbers are revealed. Leaving this information unspecified leads to an attack where the attacker launders the entire credit card number by asking to reveal different digits under different purchases.

This article does *not* propose any new declassification mechanisms. Instead we focus on the variety of *definitions* of security, in a language-based setting, which employ some form of declassification. We do not study specific proof methods, program logics, types systems or other static analysis methods. The contributions of this article are twofold:

Firstly, we provide a road map of the main declassification definitions in current language-based security research (as a timely update on security policies from a survey on language-based information-flow security [SM03]). We classify the basic declassification goals according to four axes: *what* information is released, *who* releases information, *where* in the system information is released, and *when* information can be released. Our classification includes attempts to outline connections between hitherto unrelated methods, as well as mark some clear distinctions, seeking to crystallise the security assurance provided by some known approaches.

Secondly, we identify some common *semantic principles* for declassification mechanisms:

- *semantic consistency* , which states that security definitions should be invariant under equivalence-preserving transformations;
- *conservativity*, which states that the definition of security should be a weakening of noninterference;
- *monotonicity of release*, which states that adding declassification annotations cannot make a secure program become insecure. Roughly speaking: the more you choose to declassify, the weaker the security guarantee. and
- *non-occlusion*, which states that the presence of declassifications cannot mask other covert information leaks.

These principles help shed light on existing approaches and should also serve as useful "sanity checks" for emerging models.

2. Dimensions of declassification

This section provides a classification of the basic declassification goals according to four axes: *what* information is released, *who* releases information, *where* in the system information is released, and *when* information can be released.

2.1. What

Partial, or selective, information flow policies [Coh77, Coh78, JL00, SS01, GM04, GM05] regulate *what* information may be released. Partial release guarantees that only a part of a secret is released to a public domain. Partial release can be specified in terms of precisely which parts of the secret are released, or more abstractly as a pure *quantity*. This is useful, for example, when partial information about a credit card number or a social security number is used for logging.

The PER model of information A number of partial information flow policies can be uniformly expressed by using equivalence relations to model attacker knowledge (or, perhaps more precisely, to model attacker uncertainty). Here we outline this idea (and where it has been used previously), before we go on to show how some recent approaches to declassification can be understood in these terms.

Suppose that the values of a particular secret range over `int`, and that the value of the secret is not fixed—it is a parameter of the system. Without fixing a particular value for the secret, one way to describe how much an attacker knows (or can learn) about the secret is in terms of an equivalence relation. In this approach an attacker's knowledge about the secret is modelled in terms of the attacker's ability to distinguish elements of `int`. If the attacker knows nothing about the secret then this corresponds to saying that, from the attacker's viewpoint, any value in `int` looks the same as any other value. This is captured by the equivalence relation All satisfying $\forall m, n \in \texttt{int}, m\ All\ n$. I.e., all values (or variations) of the secret look the same to the attacker. Knowledge about the secret can be modelled by other, finer, equivalence relations. For example, if the parity of a secret is to be released (and nothing else about the secret), then this knowledge corresponds to a partition of the domain into the even and the odd integers, i.e., the relation $Parity$ satisfying:

$$m\ Parity\ n \iff m \,\texttt{mod}\, 2 = n \,\texttt{mod}\, 2$$

Thus an attacker cannot distinguish any two elements in the same equivalence class of $Parity$, because at most the parity is known. At the other extreme, total knowledge of the secret corresponds to the identity relation Id.

This model of information extends to a model of information flow by describing how systems transform equivalence relations. As shown in [SS01], this is equivalent to Cohen's *selective dependency* [Coh77, Coh78], and is related to the so-called *unwinding conditions* known from Goguen and Messeguer's work on noninterference and its descendants [GM82, GM84].

It is worth remarking that noninterference in this paper is mostly concerned with protecting the secret (*high*) part of memory from the attacker who can observe the public (*low*) part of memory. This view follows the data protection view of noninterference, as it is often used in language based security [Coh78, VSI96, SM03]. This is somewhat different from the interpretation of noninterference in event-based systems that is concerned with protecting the occurrence of secret events from public-level observers [FG95, Man00, Rya01]. For the relation between these views, see [MS03, FRS05].

Suppose that we wish to express that a system leaks no more than the parity of a given secret, then we assume that the attacker already knows the parity, and show that nothing *more* is learned. This is expressed by saying that if we have any two possible values of the secret, m and n, such that $m \ \mathit{Parity} \ n$, then the attacker-observable results of running the system will be identical for these secrets. More precisely, if $s : \mathtt{int} \to \mathtt{int}$ models, for particular public inputs, how the system maps the value of the secret to the observable output, then we write $s : \mathit{Parity} \Rightarrow \mathit{Id}$, meaning that

$$\forall m, n. m \ \mathit{Parity} \ n \implies s(m) \ \mathit{Id} \ s(n)$$

In this notation standard noninterference (zero information flow) property corresponds to $s : \mathit{All} \Rightarrow \mathit{Id}$.

The use of explicit equivalence relations to model such dependencies appears in several places. In the security context it was first introduced by Cohen. Also in the information flow context, the mathematical properties of the lattice of equivalence relations was explored by [LR93]. Partial equivalence relations generalise the picture by dropping the reflexivity requirement. A partial equivalence relation (PER) over some domain D is just an equivalence relation on E such that $E \subseteq D$. Hunt [Hun91], inspired by work in the semantics of typed lambda calculi, introduced the use of the lattice of PERs as a general static analysis tool, and showed that they could be incorporated into a classic abstract interpretation framework. Abadi et. al. [ABHR99] (as well as Prost [Pro01]) use partial equivalence relations in essentially the same way to argue the correctness of dependency analyses. The present authors [SS01] showed how the PER model can also be extended to reason about nondeterministic and probabilistic systems, and also showed that not only the PER model generalises Cohen's framework but also other formulations such as Joshi and Leino's logical formulation, including the use of *abstract variables* [JL00].

Recently, *abstract noninterference* [GM04, GM05] has been introduced by Giacobazzi and Mastroeni. The properties expressible using *narrow abstract noninterference* are essentially the same as the (partial) equivalence relation models.[1] The relationship between the two approaches is not completely straightforward, however. On the one hand abstract noninterference is based on the general concept of a closure operator, so can also represent classic abstract interpretations. One tangible example of expressive power offered by abstract noninterference is that *disjunctive properties* can be expressed. For example one can express that at most *one* of secrets A and B are leaked. In the PER model, this can only be approximated by saying that both are leaked. On the other hand the use of partiality in the PER setting – useful for example in describing security properties of higher-order functions, as well as security properties of the system as a whole – cannot be directly represented in the abstract noninterference setting. For non-disjunctive properties, Clark et al. [CHM04] investigate more concretely how abstract noninterference relates to the PER model, and notably how nondeterminism is needed to model the notion of *weak observers*—observers who cannot see all the low values in the system.

The abstract noninterference framework allows for the derivation of the most powerful attacker model for which a given program is secure. This is achieved by either hiding public output data (as little as possible) [GM04] or by revealing secret input data (also as little as possible) [GM05]. Both cases contribute to the attacker's model of data indistinguishability: the former is concerned with the indistinguishability of the output while the latter treats the indistinguishability of the input. Yet Giacobazzi and Mastroeni refer to the former as "attacker models" and to the latter—somewhat surprisingly—as "declassification". The latter is claimed to be adopted from robust declassification [ZM01] by removing active attackers [GM04, GM05]. This classification does not agree with ours because, in our view, robust declassification [ZM01] addresses the question whether the declassification mechanism is robust against *active* attackers, and therefore is a "who" property (as opposed to "what"). The key property of robust declassification is that active attackers may not manipulate the system to learn more about secrets than passive attackers already know. When there are no active attackers (as in partial release), declassification is vacuously robust.

The basic idea of partial release based on (partial) equivalence relations is simple and attractive. As we shall see later, the fact that it is essentially an *extensional* definition means that it is semantically well behaved.

Related approaches In the remainder of this section we argue that two other recent approaches to declassification can be understood (at least in part) in terms of the "equivalence class" approaches—even though at first glance they appear to be of a rather different nature.

Delimited release Recent work by Sabelfeld and Myers [SM04] introduces a notion called *delimited release*. It enables the declassification policy to be expressed in terms of a collection of *escape hatch* expressions which are marked within the program via a `declassify` annotation. This policy stipulates that information may only be released through escape hatches and no additional information is leaked.

More precisely stated, a program satisfies delimited re-

1 Specifically, [GM04] notes "the PER model can easily be adapted to cope with the weaker notion of abstract noninterference."

lease if it has the following property: for any initial memory state s and any state t obtained by varying a secret part of s, if the value of all escape-hatch expressions is the same in both s and t, then the publicly observable effect of running the program in state s and t will be the same.

Interestingly, this definition does not demand that the information is *actually* released via the declassify expressions (even though the specific type system does indeed enforce this)—only that the declassify expressions within the program form the policy. As a rather extreme example, consider

$$\texttt{if true then}\ l := h * h\ \texttt{else}\ l := \texttt{declassify}(h * h)$$

This satisfies delimited release: the secret characterised by the expression $h * h$ is released to the public variable l, i.e., if we have two memories which differ only in h and for which the respective values of $h * h$ are equal, then the respective final values of l after running the above program will also be equal. This example illustrates that even when we know *what* information is released, it may be useful to also know *where* it is released.

To see how delimited release relates to the PER model, we note that every expression (or collection of expressions) over variables in some memory state induces an equivalence relation on the state. For any expression e, let $[\![e]\!]$ denote the corresponding (partial) function from states to some domain of values. Let $\langle e\rangle$ denote the equivalence relation on states $(s, t, \ldots)$ induced by $[\![e]\!]$ as follows:

$$s\langle e\rangle t \iff [\![e]\!]s = [\![e]\!]t$$

We generalise this to a set of expressions E as

$$s\langle E\rangle t \iff \forall e \in E.[\![e]\!]s = [\![e]\!]t$$

If we restrict ourselves to the two-point security lattice (for simplicity) the delimited release property can be expressed in the PER model as:

> If E is the set of declassify expressions in the program, then for all memory states s and t such that the low parts of s and t are equal, and such that $s\langle E\rangle t$, then the respective low observable parts of the output of running the program on s and t are equal.

Sabelfeld and Myers also point out that delimited release is more general than Cohen's simple equivalence relation view: declassification expressions may combine both high and low parts of the state. This provides a form of *conditional release*.[2] For example, to express the policy: "declassify h only when the initial value of l is non-zero" we can just use the expression $\texttt{declassify}(h * l)$ since from this the low observer can always reconstruct the value of h—except when l is zero. A more general form of conditional release is specified by expressions such as

$$\texttt{if}\ (\mathit{payment} > \mathit{threshold})\ \texttt{then}\ \mathit{topsecret}\ \texttt{else}\ \mathit{secret}$$

with the guarantee that the sensitive information stored in $\mathit{topsecret}$ is released if the value of the public variable $\mathit{payment}$ is greater than some constant $\mathit{threshold}$. Otherwise, the less sensitive information secret is released.

Reflecting this idea back into the equivalence relation view, this just corresponds to the class of equivalence relations which are expressed in terms of the whole state and not just the high part of the state.

2 The conditional noninterference notion from [GM84] is a predecessor to the notion of intransitive noninterference discussed in Section 2.3 on "where" definitions.

Relaxed noninterference Li and Zdancewic [LZ05] express downgrading policies by labelling subprograms with sets of lambda-terms which specify how an integer can be leaked.[3] They show that these labels form a lattice based on the amount of information that they leak. This is claimed to be closely related to *intransitive noninterference* (discussed below). Here, however, we argue that the lattice of labels from [LZ05] is closely related to the lattice of equivalence relations, and thus the class of declassification properties that can be expressed is similar to the equivalence-relation class. The semantic interpretation of a label l [LZ05][Def. 4.2.1] is closure operation:

$$\{g' \mid g' \equiv g \circ f,\ f \in l\}$$

Intuitively we can think of the meaning of a given $f \in l$ as an abstraction of all possible ways in which a program might use the result of the "leaky component" f.

In [LZ05] the equality relation ($\equiv$) in the above definition is taken to be a particular decidable *syntactic* equivalence. We will refer to this original definition as an *intentional* interpretation of labels. To relate labels to the PER model we consider an *extensional interpretation* of labels in which $\equiv$ is taken to be semantic (extensional) equality.

We can map labels to equivalence relations (over the domain of secrets) using the same mapping as for delimited release: if l is a set of lambda-terms, each with an integer-typed argument, then we define the equivalence relation on integers as:

$$m\langle l\rangle n \iff \forall f \in l.fm = fn$$

Without going into a detailed argument, we claim that the extensional semantic interpretation of labels yields a sublattice of the lattice of equivalence relations. Note in particular (as with the intentional definition) that the top and bottom points in the lattice of labels are $H \equiv \{\lambda x : \texttt{int}.c\}$

3 We focus here on the so-called *local* policies.

(for some constant c), and $L \equiv \{\lambda x : \texttt{int}.x\}$, which following the construction above can easily be seen to yield the largest equivalence relation (All, which relates everything to everything), and the smallest (the identity relation Id), respectively.

Downgrading is specified by *actions* of the form $l_1 \overset{a}{\leadsto} l_2$. In the equivalence relation view this corresponds to saying, roughly, that the action a maps arguments related by $\langle l_1 \rangle$ to results related by $\langle l_2 \rangle$, or, using the PER notation from [Hun91, SS01]:

$$a : \langle l_1 \rangle \Rightarrow \langle l_2 \rangle$$

The intentional interpretation makes finer distinctions than the equivalence relation interpretation, and these distinctions are motivated by the requirement to express not only *what* is released, but to provide some control of *how* information is leaked. Take for example the policy consisting of the single function $\lambda x.\lambda y.x == y$. In principle this function can reveal everything about a given secret first argument, via suitably chosen applications. In the extensional interpretation this policy is therefore equivalent to the policy represented by $\lambda x.x$. However, the intentional interpretation of labels distinguishes these policies – the intuition being that it is much harder (slower) to leak information using the first function than using the second.

In conclusion we see that relaxed noninterference can be understood in terms of PERS under an extensional interpretation, but provides potentially more information with an intentional interpretation. What is missing in this characterisation is a clear semantic motivation for *which* intentional equivalence is appropriate, and what general guarantees it provides. One suggestion[4] is to use a complexity preserving subset of extensional equivalence. This should guarantee that the attacker cannot leak secrets faster than if he literally used the policy functions. However it should be noted that the syntactic equivalence from [LZ05] (which include, amongst other things, call-by-name β-equivalence) is *not* complexity-preserving for the call-by-value computation model used therein. See [SGM02] for an exploration of complexity preservation issues.

Quantitative abstractions Under the category "what" we also include properties which are abstractions of "what." One extreme abstraction is to consider the *quantity* of information released. Thus we consider "how much" to be an abstraction of "what." The most direct representation of this idea is perhaps the information-theoretic approach by Clark et al [CHM02], which aims to express leakage in terms of an upper bound on the number of information-theoretic bits. The approach of Lowe [Low02] can be thought of as an approximation of this in which we assume the worst-case distribution. With this approximation the measure corresponds, roughly, to counting the number of equivalence classes in an equivalence-relation model. For a framework that integrates attacker belief into the analysis of quantitative information flow in a language-based setting see recent work by Clarkson et al. [CMS05].

4 [Peng Li, personal communication]

2.2. Who

It is essential to specify *who* controls information release in a computing system. Ignoring the issue of control opens up attacks where the attacker "hijacks" release mechanisms to launder secret information. Myers and Liskov's *decentralized label model* [ML00] offers security labels with explicit ownership information (see, e.g., [FSBJ97, PC00, BN05] for further ways of combining information flow and access control). According to this approach, information release of some data is safe if it is performed by the owner who is explicitly recorded in the data security label. This model has been used for enhancing Java with information flow controls [Mye99] and has been implemented in the Jif compiler [MZZ+04].

The key concern about ownership-based models in general is assurance that information release cannot be abused by attackers. As a step to offer such an assurance, Zdancewic and Myers have proposed *robust declassification* [ZM01] which guarantees that if a passive attacker may not distinguish between two memories where the secret part is altered then no active attacker may distinguish between these memories.

Recent work by Myers et al. [MSZ04a] connects ownership-based security labels and robust declassification by treating ownership information as *integrity* information in the data security labels. In this interpretation of robust declassification, information release is safe whenever no change in the attacker-controlled code may extract additional information about secrets. Furthermore, *qualified robustness* is introduced, which provides the attacker with a limited ability to affect what information may be released by programs. Dually to declassification, an `endorse` primitive is used for upgrading the integrity of data. Once data is endorsed to be trusted, it can be used in decisions on what may be declassified. Qualified robustness intentionally disregards the values of endorsed expressions by considering arbitrary values to be possible outcomes of endorsement.

Tse and Zdancewic also take the decentralized label model as a starting point. They suggest expressing ownership relations via subtyping in a monadic calculus and show that typable programs satisfy two weakened versions of noninterference: *conditioned noninterference* and *certified noninterference* [TZ05].

2.3. Where

Where in a system information is released is an important aspect of information release. By delegating particular parts of the system to release information, one can ensure that no other (potentially untrusted) part can release further information.

Considering *where* information is released, we identify two principal forms of locality:

Level locality policies describing where information may flow relative to the security levels of the system, and

Code locality policies describing where physically in the code information may leak.

The common approach to expressing the level locality policies is *intransitive noninterference* [Rus92, Pin95, RG99, Man01]. Recall that confidentiality policies in the absence of information release are often regulated by conventional *noninterference* [GM82, VSI96], which means that public output data may not depend on (or interfere with) secret input data. However, noninterference is over-restrictive for programs with intentional information release (average salary, information purchase, and password checking programs are flatly rejected by noninterference). Intransitive noninterference is a flow-control mechanism which controls the path of information flow with respect to the various security levels of the system. For standard noninterference the policy is that information may flow from lower to higher security levels in a partial order (usually a lattice) of security levels. The partial order relation between levels $x \leq y$ means that information may freely flow from level x to level y, and that an observer at level y can see information at level x. Since the flow relation $\leq$ is a partial order it is thus always transitive. Intransitive noninterference allows more general flow policies, and in particular flow relations which are not transitive. The canonical example (but not the only use of intransitive noninterference) is the policy that says that information may flow from *low* to *high*, and from *high* to a *declassifier* level, and from the *declassifier* level to *low*, but *not* directly from *high* to *low*. The definition of intransitive noninterference must ensure that all the downgraded information indeed passes through the declassifier, and is thereby controlled.

Mantel [Man01] introduced a variant of this idea which separates the flow policy into two parts: a standard flow lattice ($\leq$), together an intransitive downgrading relation ($\rightsquigarrow$) for exceptions to the standard flow. This has been adapted to a language-based setting by Mantel and Sands [MS04], in which both kinds of locality are addressed: intransitive flows at the lattice level, associated to specific downgrading points in the code.

Code locality can be thought of as a simple instance of intransitive noninterference based loosely on the idea of all "leaks" passing through a declassification level. Roughly speaking, the declassification constructs in the code can be thought of as the sole place where information should violate the standard flow policy. Thus the definition of intransitive noninterference should ensure that the only information release in the system passes through the intended declassifications and nothing more. With this simple view, the approaches of Ryan and Schneider (so called *constrained noninterference* [RS99]), Mullins[5] [Mul00], Bossi et al. [BPR04], and Echahed and Prost [EP05] could also be seen as forms of intransitive noninterference.

Most recently, Almeida Matos and Boudol [AB05] define a notion of *non-disclosure*. They introduce an elegant language construct `flow` F `in` M to allow the current flow policy to be extended with flows F during the computation of M. To define the semantics of *non-disclosure* the operational semantics is extended with policy labels. This could be seen as a generalisation of the "default base-policy + downgrading transitions" found in Mantel and Sands' work. Although developed independently, the bisimulation-based definition of security is very close to that of Mantel and Sands, albeit less fine grained and focused purely on code locality.

2.4. When

The fourth dimension of declassification is the temporal dimension, pertaining to *when* information is released. We identify three broad classes of temporal release specification:

Time-complexity based Information will not be released until, at the earliest, after a certain time. Time is an asymptotic notion typically relative to the size of the secret.

Examples of this category include Volpano and Smith's relative secrecy and one-way functions [VS00, Vol00]. In these cases the security definition says that the attacker cannot learn the (entire) value of a secret of size n in polynomial time. Putting it another way, the secret may be leaked only after non-polynomial time. This is related to the approaches found in Laud's work [Lau01, Lau03] and Mitchell et. al.'s work on polynomial-time process calculus (see, e.g., [LMMS98, Mit01]). Here the attacker is explicitly given only polynomial computational power, and under these assumptions the system satisfies a noninterference property.

5 Despite the similarity in terminology, there is no tight relation between Mullins' admissible interference [Mul00] and Dam and Giambiagi's admissibility [DG00]; in fact the two conditions emphasise different dimensions of declassification.

Probabilistic With probabilistic considerations one can talk about the probability of a leak being very small. This aspect is also included in Mitchell et. al.'s work, and complexity-theoretic, probabilistic and intransitive noninterference are combined in recent work of Backes and Pfitzmann [BP03]. The notion of approximate noninterference from [DHW02] is more purely probabilistic: a system is secure if the chance of an attacker making distinctions in the values of secrets is smaller than some constant ϵ. We view this (arguably) as a temporal declassification since it essentially captures the fact that secrets are revealed *infrequently*.

Relative A non-quantitative temporal abstraction involves relating the time at which downgrading may occur to other actions in the system. For example: "downgrading of a software key may occur after confirmation of payment has been received."

The work of Giambiagi and Dam [GD03] focuses on the correct implementation of security protocols. Here the goal is not to prove a noninterference property of the protocol, but to use the components of the protocol description as a specification of *what* and *when* information may be released. The idea underlying the definition of security in this setting is *admissibility* [DG00]. Admissibility is based on invariance of the system under systematic permutations of secrets.

Chong and Myers' security policies [CM04] address *when* information is released. This is achieved by annotating variables with types of the form $\ell_0 \overset{c_1}{\rightsquigarrow} \cdots \overset{c_k}{\rightsquigarrow} \underline{\ell_k}$, which intuitively means that a variable with such an annotation may be subsequently declassified to the levels $\ell_1, \ldots, \ell_k$, and that the conditions $c_1, \ldots, c_k$ will hold at the execution of the corresponding declassification points. An example of a possible domain of conditions is predicates on the variables in the program.

3. Some principles for declassification

In addressing the issue of what constitutes a satisfactory information release policy, it is crucial to adequately represent the attacker model against which the system is protected. A highly desired goal is a declassification framework that allows for modelling the different aspects of attackers, enabling the system designer to tune the level of protection against each of the dimensions. As a first step toward such a framework, we suggest some principles for declassification intended to serve as sanity checks for existing and emerging declassification models.

This section discusses the semantic consistency, conservativity, monotonicity of release, non-occlusion, and trailing attack principles. For convenience, a partial mapping of these principles to some of the models from the literature is collected in Figure 1.

3.1. Semantic consistency

This principle has its roots in the full abstraction problem [Plo77, Mit93], which has important implications for computer security [Aba98]. Full abstraction is about preserving equivalence by translation from one language to another. Viewing equivalence as the attacker's view (cf. Section 2) of the system, *semantic consistency* ensures that the view (and hence the security of the system) is preserved whenever some subprogram c is replaced by a semantically equivalent program d (where neither c nor d contains declassification). Hence, the first principle:

SEMANTIC CONSISTENCY

> The (in)security of a program is invariant under semantics-preserving transformations of declassification-free subprograms.

This principle aids in modular design for secure systems. It allows for independent modification of parts of the system with no information release, as long as these modifications are semantics-preserving. A possible extension of this principle would be one that also allows modification of code with information release, as long as new code does not release more information.

We inspect each of the release dimensions and list some approaches that satisfy this principle (and some that do not).

What Models capturing what is released are generally semantically consistent. Because what is released is described in terms of program semantics, changing subprograms by semantically equivalent ones does not make a difference from the security definition's point of view. This argument applies to partial release [Coh78, JL00, SS01, GM04, GM05], delimited release [SM04][6]. Relaxed noninterference [LZ05] aims to provide more than just "what" properties, and does so through the use of a decidable notion of equivalence (as discussed in section 2.1). Thus semantic consistency fails when we transform outside of this relation.

Who The attacker's view in the robust declassification specification [MSZ04a] is defined by low-level indistinguishability of traces up to high-stuttering (traces must agree on the sequence of assignments to low variables). Assuming that semantic equality implies low-level indistinguishability, the end-to-end

6 As elsewhere, we state this without proof (and hence with due reservations), as proofs would require a lot of detail to be given.

Property	Semantic consistency	Conservativity	Monotonicity of release	Non-occlusion
What				
Partial release [Coh78, JL00, SS01, GM04, GM05]	✓	✓	N/A	✓
Delimited release [SM04]	✓	✓	✓	✓
Relaxed noninterference [LZ05]	×	✓	✓	✓
Naive release	✓	✓	✓	×
Who				
Robust declassification [MSZ04a]	✓*	✓	✓	✓
Qualified robust declassification [MSZ04a]	✓*	✓	×	×
Where				
Intransitive noninterference [MS04]	✓*	✓	×	✓
When				
Admissibility [DG00, GD03]	×	✓	×	✓
Noninterference "until" [CM04]	×	×	✓	✓
Typeless noninterference "until"	✓*	✓	×	×

* Semantic anomalies

Figure 1. Checking principles of declassification.

nature of robustness ensures that exchanging semantically equivalent subprograms may not affect program security. This argument extends to qualified robustness [MSZ04a]. This characterisation is insensitive to syntactic variations of subprograms that are free of declassification as long as their semantics are preserved (which, for example, means that reachability is not affected).

Where The language-based intransitive noninterference condition of Mantel and Sands [MS04] satisfies semantic consistency. The definition of intransitive noninterference is built on top of a notion of k-bisimulation, where k is a security level. The basic idea is that when a declassification step occurs between two levels l and m, then (i) nothing other than that visible at level l is released, and (ii) it is only visible to the observer at level k if $m \leq k$ (i.e. k is authorised to see information at level m). After each step the bisimulation definition (following [SS00]) requires the program parts of the configurations be again bisimilar *in all states*. This is a form of "policy reset", and the same approach is adopted in *non-disclosure* [AB05].

Bossi et al's condition [BPR04] is described in an extensional way which strongly suggests that it also satisfies semantic consistency. Echahed and Prost's condition [EP05] is based on a rather unusual general computational model (a mixture of term-rewriting, constraint, and concurrent declarative programming) which makes it more difficult to assess.

When The semantic consistency principle critically depends on the underlying semantics. For complexity-sensitive security definitions [VS00, Vol00, Lau01, Lau03, LMMS98, Mit01], semantic consistency requires complexity-preserving transformations. Otherwise, for example, a program which cannot leak in polynomial time could be sped up by a transformation that compromises security.

Dam and Giambiagi's admissibility [DG00, GD03] does not satisfy the semantic consistency principle. Due to the syntactic nature of admissibility, it is possible to replace functions in the declassification protocol with semantically equivalent ones so that admissibility is not preserved. Take policy P that only allows leaking secrets via function f (which could be, for example, an encryption function). A program $S = f(h)$ is then admissible with respect to P. However, suppose the semantics of f is the identity function. Changing the program S by a semantically equivalent program $S' = h$ results in a program that is not admissible with respect to P. Notice that this is an instance of a general phenomenon: syntactic definitions of security are bound to violate the semantic robustness property. In the case of admissibility, recent unpublished results [DG03] introduce semantic information into admissibility policies via *flow automata*. This appears to be a useful feature for recovering semantic consistency.

Chong and Myers' *noninterference "until"* [CM04] is somewhat different, in that they first define a base security type-system for a mini ML-like language for handling noninterference, and *then* define the intended security condition, but only over terms which are typable according to the base security type system.

No security definition which demands that terms are typed according to a security type system (or any other computable analysis) can satisfy the semantic consistency principle with respect to *all* programs. This follows from a simple computability argument: semantic equivalence is typically not recursively enumerable, but the set of "typable" programs is either recursive or at the very least recursively enumerable. Thus there are pairs of equivalent terms for which one is typable and the other is untypable. An untypable term, according to such a definition, cannot be considered secure.

Definitions which depend on such specific analyses are not entirely satisfactory as semantic definitions. For example, a program such as

$$\texttt{if}\ h > h\ \texttt{then}\ l := 0$$

would be considered insecure by the definition since the type system makes the usual coarse approximations. In order to recover semantic consistency, the obvious fix is to lift the restriction that the definition applies to well-typed programs. Along with our analysis of the noninterference "until" definition from [CM04], presently we also explore the consequences of this generalised definition, which we refer to as *typeless noninterference "until"*. Most recently, a similar generalisation has been defined by Chong and Myers in order to combine declassification and so-called *erasure policies* [CM05].

Semantic anomalies The notion of semantic consistency is, of course, dependent on the underlying semantic model. The base-line semantic model can be thought of as defining the attacker's intrinsic observational ability. In many cases the base-line semantics is not given explicitly. However, the definition of security is often built from a notion of program equivalence which takes into account security levels. In such cases it is natural to induce the "base-line" semantics—the attacker's observational power—by considering the notion of equivalence obtained by assuming the degenerate one-point security lattice.

Having done this we can observe whether the induced semantics is a "standard" one. In the case that it is nonstandard we call this a *semantic anomaly*, which reflects something about our implicit attacker model. The presence of semantic anomalies means that semantic consistency only holds for that specific semantics.

We noted that the complexity-based definitions naturally require that the semantic model which preserves complexity—i.e., that the notion of equivalence must take into account computational complexity. This is perhaps not standard, but rather natural in this setting. We point out several examples of clear semantic anomalies, coming from Myers et al.'s robust declassification [MSZ04a], Mantel and Sands' version of intransitive noninterference [MS04], Almeida Matos and Boudol's non-disclosure property [AB05], and the typeless variant of noninterference "until." Each of these has an attacker model which turns out to be stronger than strictly necessary, and thus only satisfy semantic consistency under a stronger than usual semantic model.

- In the case of [MSZ04a] the semantic model allows the attacker to observe the sequence of low assignments (up to stuttering). This means that for any low variable l, the command $l := l$ is considered semantically distinct from `skip`, since the former contains a low assignment and the latter does not.
- In the case of [MS04], the semantic model is explicitly given as a strong bisimulation, so that only computations which proceed at the same speed are considered equivalent. It is argued, with reference to [SS00], that this allows for useful attacker models in which the attacker controls the thread scheduler. However, with a courser base-line model it is not immediately clear what an appropriate definition of intransitive noninterference should be.
- The implicit semantics of [AB05] is arguably most anomalous of those considered here. This is due to the fact that their language contains local variables. These do not fit well with the stateless bisimulation-based notion of equivalence that is induced. Consider the following example, using the usual two level base policy:

  $$\begin{array}{l}\texttt{let}\ u_L = \texttt{ref false in}\\ \quad \texttt{if}\ !u_L\ \texttt{then}\ v_L := w_H\end{array}$$

  where $!u_L$ returns the value that the reference u_L points to. This program is considered insecure, because the semantics induced by the notion of bisimulation assumes that the insecure command $v_L := w_H$ is reachable, even though it is clearly unreachable in any context. Here the anomaly cannot be described in terms of what the attacker can *observe*, but rather as an implicit assumption that the attacker can even *modify* the value of local variables. This is clearly stronger than necessary.
- For typeless noninterference "until", there are technical issues with the definition of noninterference which

assumes that an attacker can distinguish what would be normally considered equivalent functions. As a result, the program below is considered insecure:

$$\texttt{if}\ x_h\ \texttt{then}\ \texttt{ref}(\lambda x.x+1)\ \texttt{else}\ \texttt{ref}(\lambda x.1+x)$$

This could be seen as a failure of semantic consistency, since transforming $1 + x$ into $x + 1$ would make the program secure. However, we view this problem as a minor technical artifact that could be fixed for example by only allowing the attacker to observe stored values of ground type.

3.2. Monotonicity of security

Declassification effectively creates a "hole" in the security policy. For a given security definition, a program such as $l := h$ might well be considered insecure, but adding a declassification to get $l := \texttt{declassify}(h)$ and the program may well be considered secure. So the natural starting point—the base-line policy—is that absence of declassification in a program or policy implies that the program should have no insecure information flows. On the other hand, the more declassifications that a program contains, the weaker the overall security guarantee.

This leads to two related principles: *conservativity*, which says that security conservatively extends the notion of security for a language without declassification, and *monotonicity of release* based on the monotonicity of security with respect to increase in declassification annotations in code (the more declassifications you have the more "secure" you become). Let us consider each of these principles in turn.

It is sensible to require that programs with no declassification satisfy a standard security property that allows no secret leaks, as commonly expressed by some form of noninterference [GM82]. As before, we use the term noninterference to refer to the standard zero information flow policy for the language. We arrive at a principle that requires declassification policies to be conservative extensions of noninterference:

CONSERVATIVITY

Security for programs with no declassification is equivalent to noninterference.

Notice that this principle is straightforward to enforce by making it a part of security definition, which would have the flavour of "a program is secure if either it is noninterfering or it contains declassification and satisfies some information release policy." Often the conservativity principle holds trivially as it is built directly from a definition of noninterference.

Nevertheless, noninterference "until" [CM04], in the case when there is no declassification in either the policy or code, yields a strict (decidable) subset of noninterference. So, taking the same example as previously, $\texttt{if}\ h > h\ \texttt{then}\ l := 0$ is considered insecure because the type system rejects it. Thus the definition does not strictly satisfy conservativity.

Many mechanisms for declassification employ annotations to the code (or some other specification) which denote where a declassification is intended. Operationally these declassification annotations do not interfere with normal computation. At the level of annotations, the more declassify statements in a program, the weaker the overall security guarantee. This common-sense reasoning justifies the *monotonicity of release* principle:

MONOTONICITY OF RELEASE

Adding further declassifications to a secure program cannot render it insecure.

or, equivalently, an insecure program cannot be made secure by *removing* declassification annotations.

We now revisit the dimensions of release and apply the monotonicity of release principle to security characterisations along the different classes.

What Most of the examples in this class (that we have considered) express policies extensionally, so they do not rely on annotations to define the semantics of declassification. One exception is delimited release, which uses the collection of annotations in the code to determine the global policy. Adding an annotation gives the attacker more knowledge, so there is less remaining to attack. Thus if a system is secure with respect to a given degree of attacker knowledge, adding more knowledge will never make it insecure. In the PER interpretation this is just a standard monotonicity property: if a system, when presented with any two states related by some binary relation R produces equivalent observable outputs, then the same will be true when replacing R by any S such that $S \subseteq R$.

Who Declassification annotations are not used by the semantic definition of robustness [MSZ04a] (although these annotation are used in the static analysis). Therefore, program security is invariant under the removal or addition of declassification annotations. The situation is different with *qualified* robustness, however [MSZ04a]. Endorsement statements have a *scrambling* semantic interpretation that allows for arbitrary values to be the outcome of endorse. Inspired by "havoc" commands [JL00], this semantic treatment allows the difference between two values of a variable to be "forgotten" by forcing this variable to take an arbitrary value. This is justified when each endorse is preceded by a placeholder for attacker-controlled code [MSZ04b]. In this case, arbitrary values may be set to attacker-controlled variables when the control reaches the endorse. However, in general the scrambling interpretation of endorse (or declassify) might lead to the reachability of code that is otherwise dead. Dead code may mask security flaws in the program by, for instance, diverging.

It is not necessary to introduce endorsement in order to explain these reachability issues in security definitions. Consider two types of declassification, by scrambling the source and target of declassification, respectively. The intention is to "forget" the effect of declassification by requiring the source h (or target l) of a declassification operation $l := \texttt{declassify}(h)$ to take any possible value. Under the former, the semantic treatment of declassification is specified nondeterministically by

$$\langle \texttt{declassify}(h), s\rangle \longrightarrow val$$

for all val, which corresponds to the configuration with the expression $\texttt{declassify}(h)$ under some memory s that evaluates to any val. Under the latter, the scrambling is done at the result of declassification:

$$\langle l := \texttt{declassify}(h), s\rangle \longrightarrow s[l \mapsto val]$$

which does not affect the value of h but makes any value of l a possible outcome of declassification. With the respective semantic interpretations of declassify, the security condition is *possibilistic* noninterference, requiring the indistinguishability of possibilities for low-level output as high-level input is varied. We now see why both of these interpretations break the monotonicity of release. Consider the program:

$$\begin{array}{l} h := 0;\\ l := \texttt{declassify}(h);\\ \texttt{if}\ l = 42\ \texttt{then}\ l' := h' \end{array}$$

The programs is clearly secure if declassification is removed. In the presence of declassification, however, the value of l might become 42 under both scrambling semantics, and thus the insecure code $l' := h'$ becomes reachable. Hence, the program is insecure under both semantics, which contradicts the monotonicity of release.

Where The annotations of [MS04] apply to simple assignment statements. From a local perspective these would seem to satisfy the monotonicity principle, since the conditions required for a normal assignment statement are strictly stronger than for a downgrading statement. However, the fly in the ointment, as far as monotonicity is concerned, is that the definition effectively assumes that the attacker can observe the fact that a declassification operation is being performed, regardless of its content. Thus a program such as

$$\texttt{if}\ h = 42\ \texttt{then}\ \texttt{declassify}(l := l)\ \texttt{else}\ l := l$$

is insecure, because an attacker observing the presence or absence of a declassification action learns whether h was 42 or not. Removing the declassification makes the program secure.

However, in this case the fix to the definition from [MS04] seems straightforward: when nothing is leaked by the declassification then it can be viewed as a non-declassification, and vice-versa.

When Complexity-based and probabilistic declassifications are expressed extensionally, so the monotonicity principle does not apply.

For other temporal declassification conditions there is potential for monotonicity of release to fail for general reasons. Programs may contain declassifications which are in fact harmless—i.e., they do not violate noninterference, such as the declassification of a constant. But if the policy refers directly to the presence or absence of the declassification operation itself, then the very fact that a declassification statement is present in the code—albeit harmless—may cause it to violate the policy. Removing the declassify annotation might, by the same token, cause the policy to be satisfied.

By this argument, it is possible to show that admissibility [DG00, GD03] does not satisfy the monotonicity of release principle. The semantics of the protocol that specifies declassification is abstracted away, enabling harmless declassification to be disguised by the protocol's syntactic representation. Similar to semantic consistency, monotonicity of release is likely to be recovered for admissibility by introducing semantic information about declassification via flow automata [DG03].

Typeless noninterference "until" also fails monotonicity of release. One example, following the general reasoning above, is when the conditions used to trigger declassifications refer to declassifications themselves. A natural example would be a policy that says that A can be declassified if B has *not* been declassified, and vice-versa. This ensures that at most one of A and B are declassified—an interesting policy if A is the one-time pad and B is the encrypted secret. Now suppose that a program declassifies A, but contains a harmless declassification for B (i.e. one that does not *actually* reveal anything about B). The program is insecure according to the policy, but if the declassification is removed from the harmless release of B the program becomes secure. Note that monotonicity of release is preserved by noninterference "until" [CM04] because declassification conditions are simply assumed to always be guaranteed by typed programs.

3.3. Non-occlusion

Whenever declassification is possible, there is a risk of laundering secrets not intended for declassification. Laundering is possible when, for example, declassification intended only for encrypted data is applied to high plaintext. This is an instance of *occlusion*. A principle that rules out occlusion can be informally stated as follows:

NON-OCCLUSION

The presence of a declassification operation cannot mask other covert information leaks.

The absence of occlusion is a useful property of information release. Generally, declassification models along the "what" dimension satisfy this principle. Occlusion is avoided because covert flows, by the spirit of "what" models, should not increase the effect of legitimate declassification.

Occlusion is prevented in robust declassification [MSZ04a] because declassification annotations in code may not affect robustness (and hence all declassification is considered intended as long as the attacker may not affect it). Also, intransitive noninterference [MS04] successfully avoids occlusion by resetting the state (and thus the effect of each declassification), as a consequence of small-step compositional semantics.

However, qualified robustness [MSZ04a] and noninterference "until" [CM04] (when typability of programs is not required) are both subject to occlusion.

Occlusion in qualified robustness [MSZ04a] can be illustrated by occlusion in the source- and target-scrambling release definitions from Section 3.2. Indeed, slightly modifying the example we receive the program:

$$
\begin{array}{l}
h := 0; \\
l := \texttt{declassify}(h); \\
\texttt{if } l = 42 \texttt{ then loop}; \\
l' := h'
\end{array}
$$

where `loop` is an always diverging program. The program is clearly insecure because it leaks h' into l'. However, under the scrambling semantics, the value of l might become 42 after declassification, implying possible nontermination for any input state. Under termination-insensitive possibilistic noninterference (as, e.g., described by lower-powerdomain PERs [SS01]), possible nontermination under all initial memories implies security. Hence the program above is deemed secure with respect to both source- and target-scrambling definitions.

A generalisation of security policies "until" [CM04] to all programs (typeless noninterference "until") leads to occlusion. Because the security condition only considers $c_1 \ldots c_k$-free traces, i.e., traces that have not reached the last declared declassification, it is insensitive to injections of harmful code with unintended leaks after the last declassification. For instance, consider a password checking example from [CM04]:

$$
\begin{array}{l}
\textbf{int}_H\, secret := \ldots; \\
\textbf{int}_H\, pwd := \ldots; \\
\textbf{int}_H\, guess := getUserInput(); \\
\textbf{boolean}_H\, test := (guess == pwd); \\
\textbf{boolean}_L\, result := \texttt{declassify}(test, H \rightsquigarrow \underline{L}); \\
\ldots
\end{array}
$$

and inject code that leaks *secret* at level H after the password is checked. This leak is not prevented despite the fact that the security condition is variable-specific (i.e., it states noninterference "until" for each variable separately). Note that the original version of noninterference "until" [CM04] prevents such attacks by considering untyped programs as insecure from the outset. The two programs above provide evidence that the following general principle (which can be viewed as an instance of non-occlusion) is not satisfied:

TRAILING ATTACKS

Appending an insecure program with fresh variables to a secure terminating program should not result in a secure program.

One way of protecting against trailing attacks could be to introduce a special level ℓ_T that corresponds to termination and require that any declassified data be declassified to ℓ_T at the end of overall computation.

Consider a *naive release* policy that states that a program is secure if for any two runs either they preserve low equivalence of traces, or one of them executes a declassify statement. Clearly, this policy satisfies semantic consistency as semantics-preserving transformation of subprograms without declassification may not affect indistinguishability of traces for low-level observer. It also satisfies conservativity, by definition. Further, monotonicity of release also holds because the removal of declassification from an insecure program (there must exist a pair of traces without declassification that are not low-indistinguishable) may only result in an insecure program (by the same pair of traces). Although these principles hold, they are not sufficient for security assurance. This is reflected by occlusion, which naive release suffers from.

A final remark on the principles is that they are not intended to be universally necessary for all declassification policies. For example, sometimes policies are of syntactic nature by choice (as, for instance, admissibility), which makes it infeasible to guarantee appealing semantic principles. However, we argue that if one of the principles fails, it is an indication of a potential vulnerability that calls for particular attention as to why the principle can be relaxed.

4. Conclusion

Seeking to enhance understanding of declassification, we have provided a road map to the area of information release. The classification of declassification policies according to "what," "who," "where," and "when" dimensions has helped clarify connections between existing models, including the cases when these connections were not, in our opinion, made entirely accurate in the literature. For example, abstract noninterference [GM04, GM05] and relaxed noninterference [LZ05] fall under "what" models in our classification, which disagrees with connections to "who" and "where" definitions made in the respective original work. Another example is the deceptively akin admissibility [DG00] and admissible interference [Mul00] that address different dimensions of declassification ("when" and "where").

This paper is a step toward the goal of developing policies that allow combinations of policies from the individual dimensions into solid *policy perimeter defence*. Perimeter defence is a standard principle of network security: as systems are no more secure than their weakest points, they must be defended across the entire perimeter of the network. The ambition with policy perimeter is to prevent attackers from penetrating systems via weakly defended dimensions of information release.

For this to be possible we must have a better understanding of the implications of our security definitions—even more so in the presence of declassification. We have suggested some prudent principles of declassification that further help to avoid vulnerabilities in release policies, and provide tools for better understanding declassification definitions. Measuring our own previous work on declassification against these principles has revealed anomalies and "artifacts" that had previously gone unnoticed, and we suggest that the principles should serve as useful "sanity checks" for emerging models.

Acknowledgements

Thanks are due to Andrew C. Myers and Pablo Giambiagi for fruitful discussions. The paper has also benefited from the commments of Gerard Boudol, Stephen Chong, Mads Dam, Roberto Giacobazzi, Peng Li, Isabella Mastroeni, David Naumann and the anonymous reviewers.

References

[AB05] A. Almeida Matos and G. Boudol. On declassification and the non-disclosure policy. In *Proc. IEEE Computer Security Foundations Workshop*, June 2005.

[Aba98] M. Abadi. Protection in programming-language translations. In *Proc. International Colloquium on Automata, Languages and Programming*, volume 1443 of *LNCS*, pages 868–883. Springer-Verlag, July 1998.

[ABHR99] M. Abadi, A. Banerjee, N. Heintze, and J. Riecke. A core calculus of dependency. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 147–160, January 1999.

[BN05] A. Banerjee and D. A. Naumann. Stack-based access control and secure information flow. *Journal of Functional Programming*, 15(2):131–177, March 2005.

[BP03] M. Backes and B. Pfitzmann. Intransitive non-interference for cryptographic purposes. In *Proc. IEEE Symp. on Security and Privacy*, pages 140–153, May 2003.

[BPR04] A. Bossi, C. Piazza, and S. Rossi. Modelling downgrading in information flow security. In *Proc. IEEE Computer Security Foundations Workshop*, pages 187–201, June 2004.

[CHM02] D. Clark, S. Hunt, and P. Malacaria. Quantitative analysis of the leakage of confidential data. In *QAPL'01, Proc. Quantitative Aspects of Programming Languages*, volume 59 of *ENTCS*. Elsevier, 2002.

[CHM04] D. Clark, S. Hunt, and P. Malacaria. Non-interference for weak observers. In *Proc. Programming Language Interference and Dependence (PLID)*, August 2004.

[CM04] S. Chong and A. C. Myers. Security policies for downgrading. In *ACM Conference on Computer and Communications Security*, pages 198–209, October 2004.

[CM05] S. Chong and A. C. Myers. Language-based information erasure. In *Proc. IEEE Computer Security Foundations Workshop*, June 2005.

[CMS05] M. R. Clarkson, A. C. Myers, and F. B. Schneider. Belief in information flow. In *Proc. IEEE Computer Security Foundations Workshop*, June 2005.

[Coh77] E. S. Cohen. Information transmission in computational systems. *ACM SIGOPS Operating Systems Review*, 11(5):133–139, 1977.

[Coh78] E. S. Cohen. Information transmission in sequential programs. In R. A. DeMillo, D. P. Dobkin, A. K. Jones, and R. J. Lipton, editors, *Foundations of Secure Computation*, pages 297–335. Academic Press, 1978.

[DG00] M. Dam and P. Giambiagi. Confidentiality for mobile code: The case of a simple payment protocol. In *Proc. IEEE Computer Security Foundations Workshop*, pages 233–244, July 2000.

[DG03] M. Dam and P. Giambiagi. Information flow control for cryptographic applets. Presentation at the Dagstuhl Seminar on Language-Based Security, October 2003. `www.dagstuhl.de/03411/Materials/`.

[DHW02] A. Di Pierro, C. Hankin, and H. Wiklicky. Approximate non-interference. In *Proc. IEEE Computer Security Foundations Workshop*, pages 1–17, June 2002.

[EP05] R. Echahed and F. Prost. Handling declared information leakage. In *Proc. Workshop on Issues in the Theory of Security*, January 2005.

[FG95] R. Focardi and R. Gorrieri. A classification of security properties for process algebras. *J. Computer Security*, 3(1):5–33, 1995.

[FRS05] R. Focardi, S. Rossi, and A. Sabelfeld. Bridging language-based and process calculi security. In *Proc. Foundations of Software Science and Computation Structure*, volume 3441 of *LNCS*, pages 299–315. Springer-Verlag, April 2005.

[FSBJ97] E. Ferrari, P. Samarati, E. Bertino, and S. Jajodia. Providing flexibility in information flow control for object-oriented systems. In *Proc. IEEE Symp. on Security and Privacy*, pages 130–140, May 1997.

[GD03] P. Giambiagi and M. Dam. On the secure implementation of security protocols. In *Proc. European Symp. on Programming*, volume 2618 of *LNCS*, pages 144–158. Springer-Verlag, April 2003.

[GM82] J. A. Goguen and J. Meseguer. Security policies and security models. In *Proc. IEEE Symp. on Security and Privacy*, pages 11–20, April 1982.

[GM84] J. A. Goguen and J. Meseguer. Unwinding and inference control. In *Proc. IEEE Symp. on Security and Privacy*, pages 75–86, April 1984.

[GM04] R. Giacobazzi and I. Mastroeni. Abstract non-interference: Parameterizing non-interference by abstract interpretation. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 186–197, January 2004.

[GM05] R. Giacobazzi and I. Mastroeni. Adjoining declassification and attack models by abstract interpretation. In *Proc. European Symp. on Programming*, volume 3444 of *LNCS*, pages 295–310. Springer-Verlag, April 2005.

[Hun91] L. S. Hunt. *Abstract Interpretation of Functional Languages: From Theory to Practice*. PhD thesis, Department of Computing, Imperial College of Science, Technology and Medicine, 1991.

[JL00] R. Joshi and K. R. M. Leino. A semantic approach to secure information flow. *Science of Computer Programming*, 37(1–3):113–138, 2000.

[Lau01] P. Laud. Semantics and program analysis of computationally secure information flow. In *Proc. European Symp. on Programming*, volume 2028 of *LNCS*, pages 77–91. Springer-Verlag, April 2001.

[Lau03] P. Laud. Handling encryption in an analysis for secure information flow. In *Proc. European Symp. on Programming*, volume 2618 of *LNCS*, pages 159–173. Springer-Verlag, April 2003.

[LMMS98] P. Lincoln, J. C. Mitchell, M. Mitchell, and A. Scedrov. A probabilistic poly-time framework for protocol analysis. In *ACM Conference on Computer and Communications Security*, pages 112–121, November 1998.

[Low02] G. Lowe. Quantifying information flow. In *Proc. IEEE Computer Security Foundations Workshop*, pages 18–31, June 2002.

[LR93] J. Landauer and T. Redmond. A lattice of information. In *Proc. IEEE Computer Security Foundations Workshop*, pages 65–70, June 1993.

[LZ05] P. Li and S. Zdancewic. Downgrading policies and relaxed noninterference. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 158–170, January 2005.

[Man00] H. Mantel. Possibilistic definitions of security – An assembly kit –. In *Proc. IEEE Computer Security Foundations Workshop*, pages 185–199, July 2000.

[Man01] H. Mantel. Information flow control and applications—Bridging a gap. In *Proc. Formal Methods Europe*, volume 2021 of *LNCS*, pages 153–172. Springer-Verlag, March 2001.

[Mit93] J. C. Mitchell. On abstraction and the expressive power of programming languages. *Science of Computer Programming*, 212:141–163, 1993.

[Mit01] J. C. Mitchell. Probabilistic polynomial-time process calculus and security protocol analysis. In *Proc. European Symp. on Programming*, volume 2028 of *LNCS*, pages 23–29. Springer-Verlag, April 2001.

[ML00] A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology*, 9(4):410–442, 2000.

[MS03] H. Mantel and A. Sabelfeld. A unifying approach to the security of distributed and multi-threaded programs. *J. Computer Security*, 11(4):615–676, September 2003.

[MS04] H. Mantel and D. Sands. Controlled downgrading based on intransitive (non)interference. In *Proc. Asian Symp. on Programming Languages and Systems*, volume 3302 of *LNCS*, pages 129–145. Springer-Verlag, November 2004.

[MSZ04a] A. C. Myers, A. Sabelfeld, and S. Zdancewic. Enforcing robust declassification. In *Proc. IEEE Computer Security Foundations Workshop*, pages 172–186, June 2004.

[MSZ04b] A. C. Myers, A. Sabelfeld, and S. Zdancewic. Enforcing robust declassification. Submitted to *J. Computer Security*, November 2004.

[Mul00] J. Mullins. Non-deterministic admissible interference. *J. of Universal Computer Science*, 6(11):1054–1070, 2000.

[Mye99] A. C. Myers. JFlow: Practical mostly-static information flow control. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 228–241, January 1999.

[MZZ+04] A. C. Myers, L. Zheng, S. Zdancewic, S. Chong, and N. Nystrom. Jif: Java information flow. Software release. Located at `http://www.cs.cornell.edu/jif`, July 2001–2004.

[PC00] F. Pottier and S. Conchon. Information flow inference for free. In *Proc. ACM International Conference on Functional Programming*, pages 46–57, September 2000.

[Pin95] S. Pinsky. Absorbing covers and intransitive noninterference. In *Proc. IEEE Symp. on Security and Privacy*, pages 102–113, May 1995.

[Plo77] G. D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5(1):223–255, December 1977.

[Pro01] F. Prost. On the semantics of non-interference type-based analyses. In *JFLA'001, Journées Francophones des Langages Applicatifs*, January 2001.

[RG99] A. W. Roscoe and M. H. Goldsmith. What is intransitive noninterference? In *Proc. IEEE Computer Security Foundations Workshop*, pages 228–238, June 1999.

[RS99] P. Ryan and S. Schneider. Process algebra and non-interference. In *Proc. IEEE Computer Security Foundations Workshop*, pages 214–227, June 1999.

[Rus92] J. M. Rushby. Noninterference, transitivity, and channel-control security policies. Technical Report CSL-92-02, SRI International, 1992.

[Rya01] P. Ryan. Mathematical models of computer security—tutorial lectures. In R. Focardi and R. Gorrieri, editors, *Foundations of Security Analysis and Design*, volume 2171 of *LNCS*, pages 1–62. Springer-Verlag, 2001.

[SGM02] D. Sands, J. Gustavsson, and A. Moran. Lambda calculi and linear speedups. In *The essence of computation: complexity, analysis, transformation*, volume 2566 of *LNCS*, pages 60–82. Springer-Verlag, 2002.

[SM03] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, January 2003.

[SM04] A. Sabelfeld and A. C. Myers. A model for delimited information release. In *Proc. International Symp. on Software Security (ISSS'03)*, volume 3233 of *LNCS*, pages 174–191. Springer-Verlag, October 2004.

[SS00] A. Sabelfeld and D. Sands. Probabilistic noninterference for multi-threaded programs. In *Proc. IEEE Computer Security Foundations Workshop*, pages 200–214, July 2000.

[SS01] A. Sabelfeld and D. Sands. A per model of secure information flow in sequential programs. *Higher Order and Symbolic Computation*, 14(1):59–91, March 2001.

[TZ05] S. Tse and S. Zdancewic. Designing a security-typed language with certificate-based declassification. In *Proc. European Symp. on Programming*, volume 3444 of *LNCS*, pages 279–294. Springer-Verlag, April 2005.

[Vol00] D. Volpano. Secure introduction of one-way functions. In *Proc. IEEE Computer Security Foundations Workshop*, pages 246–254, July 2000.

[VS00] D. Volpano and G. Smith. Verifying secrets and relative secrecy. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 268–276, January 2000.

[VSI96] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *J. Computer Security*, 4(3):167–187, 1996.

[Zda04] S. Zdancewic. Challenges for information-flow security. In *Proc. Programming Language Interference and Dependence (PLID)*, August 2004.

[ZM01] S. Zdancewic and A. C. Myers. Robust declassification. In *Proc. IEEE Computer Security Foundations Workshop*, pages 15–23, June 2001.